TAMPEREEN TEKNILLINEN YLIOPISTO
TAMPERE UNIVERSITY OF TECHNOLOGY

TEEMU MÄKINEN

DYNAAMISTEN LUOKKIEN GENEROINTI TIETOKANNASTA

Diplomityö

Tarkastaja: Prof. Kari Systä
Tarkastaja ja aihe hyväksytty
Tieto- ja sähkötekniikan tiedekunnan
tiedekuntaneuvoston
kokouksessa 26.4.2017

TIIVISTELMÄ

TEEMU MÄKINEN: Dynaamisten luokkien generointi tietokannasta
Tampereen teknillinen yliopisto
Diplomityö, 44 sivua, 0 liitesivua
Marraskuu 2017
Tietotekniikan koulutusohjelma
Pääaine: Ohjelmistotuotanto
Tarkastajat: Prof. Kari Systä
Avainsanat: Java, tietokanta, dynaaminen, reflektio

Tämän diplomityön aiheena on uusien luokkien dynaaminen luominen ja käyttöönottaminen Java-ohjelmointikielellä, käyttäen luokkien tallennusratkaisuna tietokantaa. Lataamisen dynaamisesta luonteesta aiheutuu useita haasteita, joiden eri ratkaisuvaihtoehtoja työ käy läpi. Työ käsittelee dynaamisesti ladattavien luokkien hyötyjä ja haittoja sekä eri vaihtoehtoja näille.

Työssä tutkitaan myös ennalta tuntemattoman käyttäjien luoman koodin suorittamisesta aiheutuvia tietoturvaongelmia. Minkä tyyppisiä uhkia tästä muodostuu, sekä miten niihin voidaan varautua ja vaikutuksista selvitä.

Diplomityössä määritellään, minkälainen järjestelmä toteutettiin hyödyntämään dynaamisten luokkien lataamista tietokannasta. Osuudessa myös kerrotaan miten dynaamiset luokat hyödyttävät toteutettua järjestelmää, ja miksi toteutusvaihtoehtoihin päädyttiin.

ABSTRACT

TEEMU MÄKINEN: Generating dynamic classes from stored data in database
Tampere University of Technology
Diplomityö, 44 pages, 0 Appendix pages
November 2017
Master's Degree Programme in Information Technology
Major: Software Engineering
Examiner: Prof. Kari Systä
Keywords: Java, database, dynamic, reflection

The subject of this master's thesis is dynamic creation of new classes in the Java programming language by loading their bytecode from a database. The dynamic nature of the loading presents many challenges and the thesis investigates different methods to solve them. The thesis also considers the pros and cons of dynamically loaded classes and different options for using them.

The master's thesis explores the different security concerns rising for the system from using user defined classes. What kind of threats are formed and how they can be preempted and how to secure the system against them.

The final system using dynamic creation and loading of classes from a database is presentented in the final portition. The reasoning how dynamic classes help the implemented system and why the solution was chosen is presented.

ALKUSANAT

Haluaisin kiittää Kari Systää, joka ohjasi ja tarkasti diplomityöni. Hän oli myös kiinnostunut ja innostunut työni aiheesta, joka lisäsi myös omaa mielenkiintoani aiheeseen.

Tässä annan erikoismaininnan Juho Peltoselle, kuka suunnitteli ja kirjoitti oman diplomityönsä suostuslaskennan käyttämän sääntökielen ja kenen kanssa suositus-laskennan tekninen toteutuksen suunnittelin. Lisäksi sain apua Mikko Karttuselta parannellun LaTeX-pohjan sekä valmistumisen käytännön osuuksien selvittämisen kanssa.

Haluan myös kiittää perhettäni, erityisesti vaimoani Krista Mäkistä, joka kannusti minua eteenpäin työssäni.

Tampereen Hervannassa 24.11.2017

Teemu Mäkinen

SISÄLLYS

KUVALUETTELO

TAULUKKOLUETTELO

OHJELMALUETTELO

LYHENTEET JA MERKINNÄT

BLOB	Binary Large OBject, binäärimuotoinen tietokantatietotyyppi
Classloader	Luokka, joka lataa muut luokat Java-kielessä
JAR	Java ARchive -tiedosto
JVM	Java Virtual Machine, koodia suorittava virtuaalikone
NoSQL	Not only SQL - käsite, jolla kuvataan relaatiomallista poikkeavia tietokantoja
Sääntökieli	Suosituslaskennan sääntöjen määrittelyyn käytetty kieli
SQL	Structured Query Language - Relaatiokantojen operaatioihin luotu kyselykieli
T-LOIK	Liikenneviraston liikenteenohjausjärjestelmä

1. JOHDANTO

Perinteisien ohjelmien koodi käännetään yhden kerran, jonka jälkeen ohjelman sisäinen logiikka pysyy samankaltaisena. Kun ohjelmaa käännettäessä ei ole täysin tiedossa miten sitä tullaan ajamaan, ajon aika on mahdollista ladata ulkoisista lähteistä kirjastoja, jotka yleensä ne ovat valmiiksi tarjolla järjestelmälle. Tässä diplomityössä lähdettiin ratkaisemaan ongelmaa, miten ladata ja käyttää ohjelmaluokkia, joiden toteutus saattaa muuttua ajon aikana moneenkin kertaan. Kuinka ohjelmakoodi voidaan tallentaa tietokantaan ja ladata sieltä kun sitä tarvitaan?

Työssä käsitellään myös dynaamisten luokkien lataamiseen liittyvää ympäristöä. Mihin tarpeeseen luokkien dynaaminen lataaminen on tarvittu, ja mitä vaihtoehtoja tälle oli. Dynaamisia luokkia käytetään Liikenneviraston T-LOIK -järjestelmän ohjaussuosituslaskennan sääntöjen laskemiseen ja ohjausten tuottamiseen.

Valitun teknologian, Javan, virtuaalikoneen toimintatapaa käydään työssä läpi, sekä selvitetään, minkä takia se soveltuu tähän tarkoitukseen. Javan toiminnassa on useita hyviä puolia dynaamisen lataamisen kannalta, kuten juurikin virtuaalikone sekä tämän suomat mahdollisuudet tavukoodin kautta.

Työssä tutkitaan myös mahdollisia tallennusratkaisuita dynaamisille luokille, kuten tiedostojärjestelmää sekä eri tietokantoja.

Tämän tyyppisellä dynaamisella luokkien lataamisella annetaan ladattavalle luokalle paljon laajemmat vapaudet toimia, kuin staattiseen, ennalta määrättyä logiikkaa noudattavaan toteutukseen. Tuntemattoman koodin lataaminen ja ajaminen kuitenkin asettaa järjestelmälle tietoturvaedellytyksiä, jotta mahdollisesti pahantahtoinen koodi ei pysty aiheuttamaan vahinkoa suorittavaan järjestelmään.

Tämä diplomityö on tehty Liikenneviraston T-LOIK -järjestelmän osana olevan ohjaussuosituslaskennan laskennan toteutukseen perustuen. Projektin tämän osuuden

toteutuksesta on vastuussa Bitwise Oy. Diplomityö käsittelee järjestelmästä vain ohjaussuosituslaskennan luokkien dynaamisen lataamisen osuutta, sekä sen välitöntä ympäristöä. T-LOIK on huomattavasti suurempi kokonaisuus josta suosituslaskenta on vain pieni osa.

2. TYÖN TAUSTAA

2.1 T-LOIK

T-LOIK -järjestelmä on Liikenneviraston uusi käyttöön otettava liikenteenohjausjärjestelmä, joka korvaa käytössä olleet vanhat järjestelmät. T-LOIK yhdistää yhden käyttöliittymäkokonaisuuden alle monta eri järjestelmää, joihin kuuluu mm. ohjausjärjestelmien operointi, tilannekuvan ylläpito ja yhteydenpito sidosryhmiin. Vanha, korvattava järjestelmä ei ollut yhtenäinen kokonaisuus, vaan kokoelma erinäisiä ohjelmistoja, joita yhdessä käyttämällä liikenteen ohjaus on mahdollista.[1]

T-LOIK -järjestelmän kehittämisestä suurin tavoiteltu hyöty on, että se mahdollistaa Tieliikennekeskuksen operoinnin laadullisen kehittämisen. Järjestelmän on tarkoitus tulla tehostamaan päivystäjien kykyä seurata liikennetilanteita tuomalla kaikki saapuvat mittaukset ja tuotetut tiedot yhteen käyttöliittymään ja antamalla tietoa järjestelmän havaitsemista poikkeamista. Näin parannetaan viranomaisten yhteistä tilannekuvaa ja tehostetaan häiriöiden hallintaa.[1]

Tarkoituksena on tuottaa laadukkaampaa liikenteen ohjausta muun muassa yhdenmukaistamalla ohjausjärestelmien käyttöliittymä, tuomalla tilanteeseen toimintaohjeita operointityökalusta ja luomalla automaattiset lokikirjaukset. Parannusta tuodaan myös häiriönhallintaan, T-LOIK mahdollistaa häiriöttömien ohjausjärjestelmien operointivastuun siirtämisen toiseen keskukseen.[1]

T-LOIK:n kokonaisuuden osa jota tämä työ käsittelee on ohjaussuosituslaskenta, joka laskee erinäisten sääntöjen perusteella mitä digitaalisia liikennemerkkejä ja -opasteita ohjataan näyttämään.

2.2 Suosituslaskenta

Ympäri Suomea on olemassa teillä erilaisia sää- ja liikenneantureita, jotka mittaavat vallitsevia olosuhteita. Mitattavia suureita ovat sääasemien osalta esimerkiksi ilman- ja tien lämpötila, näkyvyys, tuulen nopeus tai sadetilanne. Liikenneasemilta anturit mittaavat muun muassa liikenteen määrää, autojen keskinopeutta ja tien varausastetta. Näiden asemien tuottamat mittaukset kulkeutuvat omiin järjestelmiinsä, josta tiedot lähetetään edelleen eteenpäin. Suosituslaskenta vastaanottaa kerättyä tietoja, ja laskee järjestelmään määriteltyjen sääntöjen perusteella miten liikennettä tulisi ohjata. Suosituslaskennan tulisi järjestelmänä pystyä toteuttamaan Liikenneviraston määrittelemiä ohjauspolitiikkoja, joiden mukaan automaattisille järjestelmille luodaan sääntöjä. [2] Järjestelmä kuitenkin tarvitsee laskennan ja sääntöjen suorittamiseen ei-triviaalia logiikkaa, joten sitä ei toteuteta käyttämällä valmiita sapluunoja, vaan suunnittelija pystyy toteuttamaan säännöt käyttäen tarkoitusta varten erikeen kehitettyä sääntökieltä.

2.3 Sääntökieli

Järjestelmään on luotu erillinen sääntökieli, jolla on mahdollista luoda aiemmin mainittuja ohjaussääntöjä. Ohjaussäännöillä ohjataan liikenteessä olevia opasteita sekä liikennemerkkejä. Säännöt saattavat olla monimutkaisiakin, mutta sääntöjen kirjoittajat eivät ole ohjelmoijia, joten sääntökieli on yksinkertaistettu täsmäkieli. Kuitenkin säännöt tarvitsevat ohjelmointikielen ominaisuuksia, kuten ehtolauseita ja muuttujia, joten ongelman ratkaisuksi sääntökieli käännetään javakoodiksi. Näin kielestä saadaan palvelimella rajatussa ympäristössä suoritettavia luokkia, jotka toimivat kuten sääntöjen kirjoittajat haluavat.[3]

Sääntökieli on toteututettu käyttäen Xtend-kieltä[4], toimien ohjelmointikielien kehitykseen tarkoitetussa Xtext-ohjelmistokehyksessä[5]. Tämä kokonaisuus lopulta toimii Eclipse-ympäristön sisällä, jonka avulla saadaan nykyaikaisen IDEn toiminnallisuus helposti käyttöön. Sääntökieli on oma kokonaisuutensa, mutta se liittyy oleellisesti tähän työhön, sillä sääntökielen avulla luodaan dynaamisesti ladattavat Java-luokat.

2.4 Ohjauspolitiikat

Säännöt, joiden mukaan laskenta suoritetaan on määritelty liikenneviraston ohjauspolitiikka-ohjeissa. [2]

Ohjauspolitiikat ovat tienpitäjän suunnittelemat kuvakset siitä, miten etäohjattavia liikenneopasteita ja kylttejä ohjataan ja millä ehdoilla. Ohjauspolitiikka määrittelee ne sää-, keli- ja liikennetilanteet ynnä muut olosuhteet, joiden perusteella opasteita ohjataan. Tämä muodostaa järjestelmän perusteet, jotka toimivat joko automaattisesti tai järjestelmän päivystäjän toimesta. Ohjauspolitiikka sisältää ohjausperusteiden määrittelyn lisäksi kuvauksen järjestelmän toimintaympäristöstä, tavoitteista ja vastuista.

Ohjauspolitiikka vaikuttaa:

- Tienkäyttäjiin ohjauspolitiikan vaikutusalueella
- Järjestelmän suunnitteluun ja kehitykseen
- Päivystäjien toimintaan tieliikennekeskuksissa

Tieliikenteen hallinnassa on käytössä useita vaihtuvia ohjausjärjestelmiä, sekä suunnitteilla on lisää. Tämä luo tarpeen yhtenäisille ohjeille, joka keskitetysti kertoo miten järjestelmien tulisi toimia, riippumatta siitä miten ne on toteutettu.

Ohjauspolitiikat eivät rajoitu vain yleiseen ohjaukseen, vaan nämä on määritelty sijainnin lisäksi myös Suomen ilmaston vuoksi myös ajallisesti. Kesä-, talvi- sekä siirtymäajat ovat olemassa erikseen, ja näiden alku- ja loppuajat ovat dynaamisia.

Esimerkiksi lainaus pienestä osasta moottoritien hyvän kelin ei-siirtymäkauden kesäajan ohjauspolitiikan sisääntuloista, ilman ohjauksia [2]:

Taulukko 2.1 *Ei-siirtymäajan kesäajan ohjauspolitiikkaa, liikenneviraston ohjauspoliittkojen laadintaohjeesta*

OLOSUHTEET		
Anturit	Parametrit	Parametrien arvot
A1) Hyvät olosuhteet – Kaikki seuraavat anturikohtaiset ehdot täyttyvät		
Tienpinta-anturit / optiset kitka- ja lämpötila-anturit	Oikean JA Vasemman kaistan keli / tila TAI Oikean TAI Vasemman kaistan keli / tila JA Tien pinnan lämpötila	1 = kuiva TAI 2 = kostea TAI 3 = märkä TAI 4 = märkä ja suolattu TAI 5 = kuura TAI 8 = tod. näk. kostea ja suolainen JA Tien pinta >= +2°C
Optiset kitka-anturit / Opt. lämpötila-anturi, tienpintaanturit	Oikean JA Vasemman kaistan kitka (mikäli myös vas. kaistalla anturi) TAI Tien pinnan lämpötila	μ >= 0,40 TAI Tien pinta >= +2°C
Sadeanturi ja oik. puol. tienpinta-anturi / opt. kitkaanturi	Sade TAI Sade JA Oikean kaistan keli	0 = pouta TAI 1 = heikko TAI TAI 2 = kohtalainen sade JA (1 = kuiva TAI 2 = kostea)

Yllä olevan kaltaisen, vielä suhteellisen yksinkertaisen ohjauspolitiikan, vaatimien sääntöjen toteuttaminen olisi todella hankalaa yksinkertaisemmalla järjestelmällä. Tämän vuoksi ohjauspolitiikka toteutetaan sääntökielellä, josta saatu tavukoodi suoritetaan palvelimella.

2.5 Adaptiivinen

Järjestelmän luonteesta johtuen sen täytyy myös olla adaptiivinen, eli sopeutua ajonaikaisiin muutoksiin. Tämän järjestelmän suoritus ja suoritettavat säännöt saattavat muuttua ohjelman ajon aikana. Ei että koko T-LOIK tarvitsee ajaa alas, jos jotain sääntöä on muutettu jossain päin Suomea. Sääntöjä täytyy olla mahdollista muuttaa kesken ajon, ilman että se vaatii järjestelmän uudelleen käynnistystä. On mahdollista, että säännöt vaativat huomattavan paljon testausta ja useita latauskertoja ennen kuin ne ovat valmiita. Tämä ei ole luokkien avulla mahdollista kaikilla

ohjelmointikielillä, mutta se oli yksi valintakriteeri, joka vaikutti päätökseen kielestä ja toteutustavasta.

2.6 Ohjelmointikielen valinta

Koska uusien sääntöjen laskeminen täytyy olla mahdollista käynnistämättä järjestelmää uudestaan, käytettävän kielen tulee tukea uusien luokkien dynaamista lisäystä. Tuki tälle on lähes jokaisessa ohjelmointikielessä, mutta on myös toimintaa helpottavaa, jos koodia ei tarvitse kääntää laitteen konekielelle. Pienimmällä vaivalla tämä onnistuu skriptikielillä, kuten esimerkiksi Pythonilla tai JavaScriptillä. Kuitenkin sääntöjä laskeva järjestelmä on toteutettu jo valmiiksi Javalla ja toimii Javan virtuaalikoneen päällä, joten se on luonnollinen valinta kieleksi. Lisäksi mikä tahansa Javan virtuaalikoneen päällä toimiva kieli osaa suorittaa näitä luokkia, mikäli järjestelmän alustaa tarvitsee jostain syystä vaihtaa. Näitä alustavaihtoja ovat esimerkiksi Clojure, Scala ja Kotlin.

Javan puolesta puhuu myös se, että käännettyjen tavukoodiluokkien ei tarvitse olla käännetty samalla, tai edes samalla arkkitehtuurilla toimivalla järjestelmällä kuin suorittavan järjestelmän. Ohjaussuosituslaskennan tapauksessa säännöt voidaan kääntää tavukoodiksi Scalalla toimivalla sääntökielieditorilla, jonka jälkeen ne voidaan lähettää laskentaa suorittavalle palvelimelle.

3. JAVA

3.1 Java-ohjelmointikielen historiaa

Java on vuonna 1995 julkaistu ohjelmointikieli, jonka alkuperäisenä tavoitteena oli luoda alusta, joka toimisi missä ympäristössä hyvänsä. Kielen luojat James Gosling, Mike Sheridan ja Patrick Naughton tähtäsivät tekevänsä kielen, jolla pystyisi kirjoittamaan koodin kerran, jonka jälkeen se toimisi alustalla millä hyvänsä. Tämä millä tahansa alustalla toimiminen toteutettiin Javan virtuaalikoneen, JVM:n avulla. Alun perin kohdealustoina olivat sulautetut järjestelmät, mutta Javan toteutus toimi hyvin myös tietokoneilla. Yksi Javan tavoitteista sen alkuaikoina oli muodostua internetin ohjelmointikieleksi, mutta siinä tavoitteessa ei onnistuttu mm. tietoturvaongelmien vuoksi. Java on olio-ohjelmointikieli, joka on ottanut hyvin paljon vaikutteita C++:sta, karsien luojiensa mielestä turhia sekä vaarallisia ominaisuuksia kuten osoittimet, bittikentät ja etumerkittömät numerot. Lähes kaikki muuttujat Javassa ovatkin olioita, lukuunottamatta numeerisia perustyyppejä int, long, float, double sekä totuusarvotyyppiä boolean. Nykyaikana tuosta perinnöstä suurimpana hyötynä on virtuaalikone, jonka ansiosta koodia ei tarvitse kääntää kohdejärjestelmälle. Oli tavukoodi luotu järjestelmällä millä hyvänsä, vaikkapa 32 bittisellä Windowsilla, se tulee toimimaan myös 64 bittisellä Linuxilla. Vaatimuksena on vain se, että alustalle on saatavilla Javan virtuaalikone.[6]

Javasta on löytynyt runsaastikin tietoturva-aukkoja vuosien varrella, mutta ne eivät vaikuta sen käyttöön palvelinympäristössä, jossa se edelleen säilyttää paikkaansa yhtenä suosituimmista valinnoista. Ongelmat ovat suurimmaksi osaksi rajoittuneet Javan selaimissa toimineisiin Java Appletteihin, jotka mahdollistivat koodin ajamisen internetin käyttäjien koneilla ilman takeita siitä, mitä koodi oikeasti teki. Hyökkääjän oli mahdolista murtautua ulos sille asetetusta hiekkalaatikosta ja päästä käsiksi käyttäjän järjestelmään. Nykyisin Java Applettien käytöstä on luovuttu

lähes kaikkialla, eivätkä selaimet suorita niitä ilman erillistä luvan antamista.

3.2 Virtuaalikone

Javan suorittamisesta vastaa Javan virtuaalikone, JVM eli Java Virtual Machine. JVM on on nimensä mukaisesti virtuaalikone, joka suorittaa ajettavan koodin omassa virtuaalisessa ympäristössään, jolloin koodi voi olla alustasta riippumatonta. Koska JVM on virtuaalikone, siinä on myös oma käskykantansa, jota suoritettavat ohjelmat käyttävät. Javan virtuaalikone ei siis ole skriptikieli, joka suorittaa suoraan ihmisen luottavaa kooditiedostoa, vaan JVM suorittaa Javan omaa tavukoodiformaattia. Tavukoodi yleisesti tallennetaan .class-tiedostoihin, mutta tämä ei sinänsä ole vaatimus vaan yleinen konventio selkeyden vuoksi. Kaikki tavukoodi haetaan ajon aikana toisin kuin esimerkiksi C++:ssa tai muissa kielissä, joissa ainoastaan erikseen kirjastoiksi merkitty koodi on mahdollista ladata ajon aikana. [6]

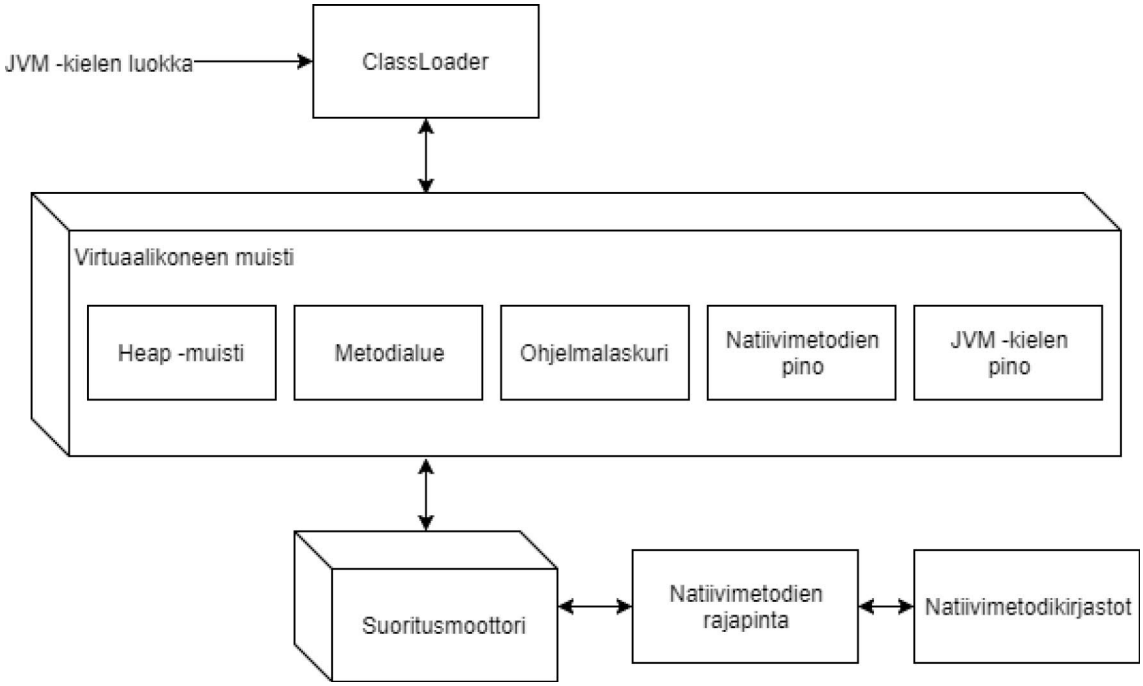
3.3 Virtuaalikoneen toimintaperiaate

Javan virtuaalikone muodostaa kuvan 3.1 mukaisen kokonaisuuden, jossa virtuaalikoneella on oma muistialueensa, jossa se toimii. Tässä muistissa on kaikki virtuaalikoneen tarvitsema, kuten ohjelmalaskuri ja pinojen muistialueet.

Kun koodissa kutsutaan metodia, se muodostaa oman ajokehyksensä, framen. Tämän kehyksen sisällä on oma ohjelmalaskurinsa, joka suorittaa ohjelmaa eteen päin; Javassa ohjelmalla ei ole mahdollista päästä käsiksi koko koko järjestelmän muistiin, jossa pystyisi esimerkiksi goto-käskyllä hyppäämään täysin eri osioon, vaan käytössä on vain virtuaalikoneelle varattu muistialue. Rekistereitä virtuaalikoneessa vastaa paikalliset muuttujat, jotka ovat myös ajokehyskohtaisia. Näitä käytetään nimensä mukaan kehyksen sisäisiin muuttujiin, esimerkiksi silmukoiden laskureissa. Javan virtuaalikoneessa voi olla vain yksi ajokehys kerrallaan aktiivisena.[6]

Kun ajokehys on valmis, se palauttaa arvon kuten metodi yleisesti palauttaisi. Myös void-tyyppinen metodi palauttaa tyhjän arvon. Palautusarvon lisäksi mahdollinen poistumistapaus on poikkeuksen heittäminen, jolloin jos poikkeusta ei käsitellä metodin sisällä, se palautetaan ulos kehyksestä. Tässä tapauksessa metodi ei myöskään palauta mitään arvoa.

Kuva 3.1 *Java Virtual Machine*



Javan virtuaalikone toimii käyttäen pinoa, johon kasataan suoritettavat operaatiot. Operaatiot suoritetaan pinosta päältä jolloin saadaan ulos suorituksen arvo. Uudet operaatiot lisätään taina pinon päälle (FIFO stack). Jokaisella suorittavalla säikeellä on oma pinonsa, jota se käsittelee. Virtuaalikoneen käskykanta erottelee operaatiotyyppejä käyttämällä erillisiä tavukoodeja eri datatyyppien toiminnoille. Käännettyyn koodiin sisältyvät ohjeet, jotka toimivat tyypitetyllä tiedoilla, ovat kaikki erikoistuneita kokonaislukutyyppiin.[6]

Virtuaalikoneen käskykanta tuntee operaatiot seuraaville muuttujatyypeille: liukuluvut float ja double ja kokonaisluvut byte, short, int, long, char. Javan virtuaalikone ei tunne totuusarvon käsitettä, vaan käyttää totuusoperaatioihin suorittamiseen kokonaislukuoperaatioita. Käskykanta tuntee myös muuttujatyypit luokka, rajapinta ja taulukko. Luokan ja rajapinnan käsitteet ovat olennainen osa Javaa, joten ei ole ihme, että ne löytyvät suoraan virtuaalikoneestakin. Suora taulukkotyyppien käsittely ei onnistu perus assemblykieleltä, johon Javan tavukoodia tässä on verrattu.[6]

Oma asiansa on virtuaalikoneen muistinkäsittely ja säikeet, mutta ne menevät tämän työn laajuuden yli.

3.4 Javakoodista tavukoodiksi

Seuraava javakoodi suorittaa ulomman silmukan 100 kertaa ja sisemmän silmukan 10 kertaa per ulomman silmukan suoritus.

```
1 int i;
  int j;
3 for (i = 0; i < 100; ++i) {
      for (j = 0; i < 10; ++j) {
5         ; // NOP
      }
7 }
```

Ohjelma 3.1 *Silmukkakoodi*

Kun tämä koodi ajetaan kääntäjän läpi, se muutetaan seuraavantyyppiseksi tavukoodiksi

```
0   iconst_0
1   istore_1
2   iinc 1 1
3   iload_1
4   bipush 100
5   if_icmpeq 14
6   iconst_0
7   istore_2
8   iinc 2 1
9   iload_2
10  bipush 10
11  if_icmplt 8
12  goto 2
13  return        // Valmis
```

Ohjelma 3.2 *Silmukkakoodi tavukoodina*

Ensin aloitetaan i-silmukan käyminen: rivillä 0 alustetaan vakio 0 pinon päälle, ja heti seuraalla rivillä sijoitetaan paikalliseen muuttujaan 1 se arvo, joka on pinon päällä, eli tässä tapuksessa 0. Rivillä 2 paikallisen muuttujan 1 arvoa kasvatetaan yhdellä. Rivillä 3 iload_1 lataa paikallisen muuttujan 1 arvon pinon päälle. Rivillä 4 asetetaan pinon päälle arvo 100, ja rivillä 5 vertaillaan kahta pinon päällä olevaa

lukua keskenään; jos ne ovat yhtä suuret hypätään määritellylle riville 14, muussa tapauksessa jatketaan suoritusta normaalisti.

Seuraavaksi käydään läpi sisempi silmukka. Rivillä 6 alustetaan taas 0 pinon päälle ja rivillä 7 tallennetaan pinon päällä oleva luku paikalliseen muuttujaan 2. Samaan tapaan paikallisen muuttujan 2 arvoa kasvatetaan yhdellä, ladataan paikallinen muuttuja 2 pinon päälle, ladataan vertailuarvo 10 pinon päälle ja vertaillaan näitä keskenään. Tässä tapauksessa käytetään pienempi kuin -vertailua, jossa katsotaan onko pinon alempi luku pienempi kuin päällimmäinen, ja jos on, niin hypätään riville 8, eli j:n arvon kasvattamiseen. Mikäli vertailu on epätosi, jatketaan suorittamista riville 12, jossa poistutaan j-silmukasta.

Heti seuraavalla rivillä hypätään i-silmukan alkuun ja kasvatetaan paikallisen muuttujan 1, eli i:n arvoa yhdellä. Kun i:n arvo on tarpeeksi suuri hypätään riville 13, jossa palautetaan mahdollinen paluuarvo.

Tavukoodi ei ole tavallisesti luettaessa yhtä selkeää lukea kuin ylle kirjoitettu esimerkki, vaan käyttää sisäisesti komentojen heksa-arvoja eikä selkeämpiä kirjoitusasuja. Tekstimuotoinen tavukoodi saadaan ajamalla tavukoodi kokoajaohjelman läpi, tämä onnistuu esimerkiksi Javan mukana tulevalla javap-ohjelmalla.

3.5 Tavukoodista konekieleksi

Vaikka tavukoodi muistuttaa paljon konekieltä, sitä ei kuitenkaan voi suoraan muuttaa jokaisen koneen ymmärtämään muotoon. Tämän vuoksi Javan virtuaalikone tuleekin olla jokaiselle alustalle erikseen käännetty. Useilla arkkitehtuureilla pino- ja aritmeettiset operaatiot onnistuvat lähes samanlaisina kuin tavukoodissa, mutta luokat, rajapinnat ja taulukot eivät toimi samalla tavalla.

Konekielet eroavat prosessorin arkkitehtuurin mukaan, ARM-prosessorin kieli eroaa x86-arkkitehtuurin kielestä huomattavasti. Jopa Intelin ja AMD:n prosessoreiden välillä on eroja konekielessä, vaikka molemmilla onkin yhteisiä komentoja.

On myös olemassa Java-kääntäjiä, jotka luovat suoraan konekielisiä tiedostoja. Yksi tämänlainen kääntäjä on GCJ, Gnu Compiler For Java. GCJ voi kääntää javakoodin tai valmiit tavukooditiedostot suoraan konekielelle.[7] GCJ:n kehitys lopetettiin

heinäkuussa 2017, ja sille ei ole tiedossa tätä diplomityötä kirjoitettaessa seuraajaa.

Ottaen huomioon kuinka hankalaa konekieli on, ja miten epäselvää sen lukeminen on perehtymättä sen toimintaperiaatteisiin syvällisesti, tässä diplomityössä ei käydä mitään esimerkkiä läpi konekielestä.

4. LUOKKIEN LUOMINEN TAVUKOODISTA

4.1 Javan Object -luokka

Kaikki Javan luokat periytyvät yksinkertaisimmasta Object-luokasta. Tämä luokka toteuttaa vain muutamia luokan perusmetodeita:

clone luo kopion oliosta

equals yhtäsuuruusvertailu toisen Objectin kanssa

finalize siivoaa olion käyttämät viitteet ja yhteyden kun roskienkeruujärjestelmä poistaa sen

getClass kertoo olion luokan

hashCode palauttaa lähes uniikin olion tarkisteen, jolla olio voidaan yksilöidä järjestelmässä

notify, notifyAll herättää säikeen, joka odottaa tätä oliota

toString olio antaa itsestään tekstimuotoisen kuvauksen tai version

wait odottaa kunnes kuluu määrätty aika tai olio saa notify-kutsun

Tämä rajapinta riittää vain luokkien perushallintaan järjestelmässä. Koska Object on hierarkiassa matalin luokan taso, se on ainoa jota voidaan käyttää kun käsitellään tuntematonta oliota.

4.2 Classloader

Luokkien lataaminen toimii Javan Classloaderien avulla. Ne vastaavat nimensä mukaisesti luokkien lataamisesta ohjelman muistiin suoritettaviksi.

Classloaderit jaetaan kahteen eri tyyppiin: Bootstrap classloader, sekä muut classloaderit. Bootstrap classloader tulee suoraan järjestelmästä ja sillä on erioikeuksia johtuen luonteestaan. Muut classloaderit jaetaan karkeasti seuraaviin tyyppeihin:

- Extension classloader
- System classloader
- Muu classloader

4.2.1 Bootstrap classloader

Bootstrap classloader on sisäänrakennettu Javan virtuaalikoneen toteutukseen ja sen vastuulla on ensimmäisten luokkien luominen. Bootstrap classloader lataa Object-toteutuksen muistiin, kuten myös Classloader-luokan, jotta uusia voidaan luoda. Bootstrap lataa myös suoritettavan ohjelman pääluokan muistiin suoritettavaksi. Tämä on ohjelman metatiedoissa määritelty luokka, jonka staattisen main-metodin suorittamalla ohjelman suoritus aloitetaan. Yleisimmin pääluokka määritellään ohjelman tietoja sisältävässä manifest-tiedostossa, mutta se voidaan myös määritellä ohjelman käynnistyskomennossa parametrina.[6]

Bootstrap classloader on siitä erikoinen, että siihen ei pysty viittaamaan enää ohjelman suorituksen aikana suoraan. Bootstrap classloaderin avulla voidaan luoda olioita vain kun muut classloaderit kutsuvat ylempää toteutusta luomaan luokkia.

Bootstrap classloader on myös siitä poikkeava, että sen toimintaan ei pysty normaalitilanteessa puuttumaan, johtuen sen kytköksistä virtuaalikoneeseen.

4.2.2 Extension classloader

Ehkä tärkein classloader bootstrapin jälkeen on yleisesti käytetty extension classloader, joka lataa suoritukseen määriteltyjen ulkoisten kirjastojen luokat. Nämä luokat haetaan Javan suoritusympäristön, Java Runtimen Evironmentin (JRE) sijainnsita jre/lib/ext. Nämä ovat järjestelmänlaajuisia kirjastoja, jotka eivät riipu suoritettavan ohjelmaan sisällytetyistä kirjastoista.[6]

4.2.3 System classloader

Toinen tärkeä on system classloader, joka hoitaa classpath-ympäristömuuttujaan määriteltyjen luokkien lataamisen. CLASSPATH on muuttuja, joka kertoo suoritukselle, mistä ohjelman tarvitsevat kirjastot löytyvät. Yleisesti nämä ovat .jar-tiedostoja, jotka suoritettava ohjelma tarvitsee, mutta eivät ole koko järjestelmän kannalta oleellisia. Tähän joukkoon kuuluvat lähes kaikki käytetyt kirjastot.[6]

4.2.4 Muut classloaderit

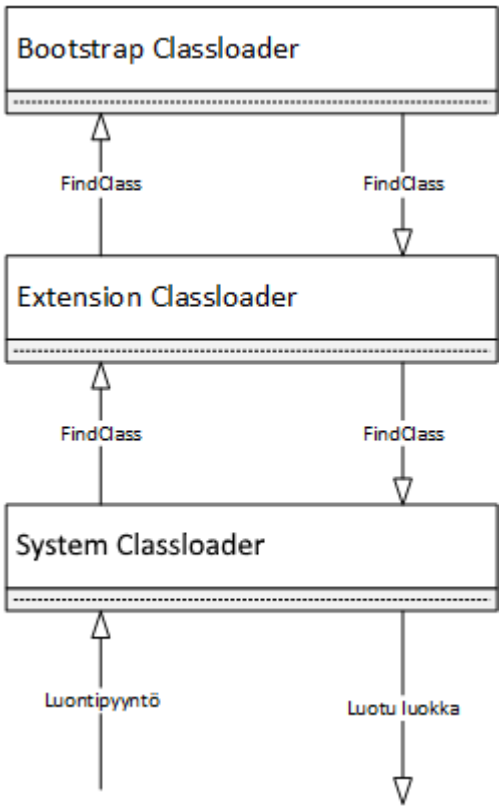
Muut classloaderit käsittävät nimensä mukaan kaikki muut. Näitä ovat esimerkiksi ohjelman määrittelemät classloaderit, jotka osaavat suorituksen aikana luoda uusia luokkia dynaamisesti annettujen parametrien perusteella. Luokkien lataamisen hierarkia toimii niin, että luokka tietää sen classloaderin, joka sen on ladannut. Koska classloaderit ovat myös olioita, niin ne tietävät myös hierarkiassa yhden ylemmän classloaderin. Tätä reittiä pitkin on mahdollista luoda minkä tahansa luokan olio, jonka hierarkiassa ylemmät classloaderit tuntevat.

4.2.5 Classloadereiden toiminta

Yleisin classloaderin toimintatapa on hakea luokat tavukoodiksi käännetyistä .class-tiedostoista, jotka on määritelty Javan virtuaalikoneen “Classpath”-muuttujaan. Classpath saattaa sisältää useita hakemistoja, joista luokkia etsitään niiden nimen perusteella. Koska luokkien nimet saattavat olla hyvinkin geneerisiä, käytetään luokkiin viittaamisessa niiden koko nimeä, joka on yleisesti muotoa `"alue.yritys.projekti.paketti.luokka"`. Esimerkiksi Javan JAR-tiedostojen sisältämien luokkien lukemiseen tarkoittetu OpenJDK:n ClassReader-luokka löytyy nimellä `"com.sun.java.util.jar.pack.Reader"`. Poikkeus tähän sääntöön on Javan oletuskirjastojen luokat, jotka alkavat vain `"java.tyyppi.luokka"`, kuten oletuskirjastoon kuuluva merkkijonototeutus on `"java.util.String"`.

Kuitenkin on mahdollista, että luokat ladataan lähes mistä tahansa lähteestä, josta voidaan vain lukea tavukoodia. On esimerkiksi olemassa javakääntäjätoteutuksia, jotka kääntävät koodin tavukoodiksi muistiin, joka voidaan tämän jälkeen ladata luokkana ohjelmaan.

Kuva 4.1 *Java classloadereiden hierarkia*



4.2.6 Classloadereiden käyttö dynaamisten luokkien kanssa

Classloaderit ovat käytännössä ainoa keino luoda uusia luokkia Javassa, mutta niiden käytössä on olemassa yksi ongelma dynaamisia luokkia luodessa; kun yksi classloader lataa luokan, se myös takaa, että kaikki samantyyppiset luokat tulevat tulevaisuudessakin olemaan samanlaisia. Tämä on ongelma, jos käytettäviä luokkia halutaan muuttaa ohjelman ajon aikana. Tämä pakottaa järjestelmää pitämään kirjaa milloin luokkien tavukoodiin tehdään muutoksia, ja tuhoamaan olemassa olevan classloaderin kun muuttunutta luokkaa oltaisiin luomassa. Tämän jälkeen luodaan uusi classloader, joka ei tunne vielä yhtään luokkaa, joten se onnistuneesti lataa uuden muuttuneen luokan määritelmän ja luo sen ongelmitta. Tämä takuu varmistetaan pitämällä perus classloaderin toteutuksessa defineClass metodin toteutus final:ina. Perus classloaderilla tarkoitetaan sitä, josta kaikki muut classloaderit peritään. Final tarkoittaa, että kyseistä metodia ei pystytä korvaamaan toisella perityissä luokissa. DefineClass on myös juuri se metodi, joka luo tavukoodista käyttökelpoisia luokkia.[8]

```
class ByteClassloader extends Classloader {

  private Map<String, Class> ladatutLuokat;

  @Overload
  public Class findClass(String luokanNimi) {
    if (ladatutLuokat.contains(luokanNimi)) {
      return ladatutLuokat.get(luokanNimi);
    }
    return lataaLuokkaTietokannasta(luokanNimi);
  }

  private Class lataaLuokkaTietokannasta(String luokanNimi) {
    ...
  }
}
```

Ohjelma 4.1 *Osa tavukoodi -classloaderin toteutuksesta*

Esimerkki manuaalisesti toteutetun classloaderin toiminnasta

4.3 Reflektio

Yksi Javan tunnetuimpia ominaisuuksia on reflektio. Reflektion avulla Javan virtuaalikoneen on mahdollista ajon aikana käsitellä luokan osia, jotka eivät muuten olisi avoimia. Esimerkiksi luokan yksityisiä muuttujia on mahdollista muuttaa tai kutsua ennalta tuntemattomia metodeita niiden nimen perusteella. Erityisesti reflektiota käytetään ohjelmien testauksessa, sillä silloin on mahdollista korvata luokkien metodeita sekä muuttujia testausta varten sopivilla mockeilla ja stubeilla. Myös vastaanotettujen parametrien lukeminen onnistuu asettamalla reflektion avulla metodiin kuuntelija väliin, jolloin voidaan varmistaa ohjelman toimintaa.

Reflektio on yksi vaihtoehto toteuttaa järjestelmän vaatima dynaaminen luokkien suoritus.

Esimerkki reflektiosta:

```
class Luokka {
  public void metodinNimi(String teksti) {
    System.out.println(teksti);
  }

  public String metodi(String teksti) {
    return teksti;
  }
}

olio = new Luokka();

Class luokka = olio.getClass();
Method metodi = luokka.getMethod("metodinNimi", String.class);
metodi.invoke(olio, "hello");
```

Ohjelma 4.2 *Esimerkki reflektion käytöstä*

Reflektio toteutetaan yllä olevassa listauksessa käyttäen Javan Reflection API:a. Koodissa kutsutaan Luokka -luokan metodinNimi -metodia antaen sille merkkijono-parametriksi "hello".

Reflektiossa on kuitenkin olemassa suorituskykyongelmia. Uudempien javaversioiden myötä suorituskykyongelmat ovat vähentyneet ja reflektion käyttö ei enää ole yhtä pahaksi suorituskyvylle, mutta silti sitä ei suositella. Testasin vertailun vuoksi olisiko reflektion käyttö tässä tapauksessa järkevä ratkaisu vai ei. Suoritin yllä olevan Luokka-koodin seuraavilla kolmella eri kutsutavalla, ja kirjasin suoritusajat ylös.

Normaali metodikutsu on `luokka.metodi()`.

```
olio.metodi("teksti");
```

Ohjelma 4.3 *Suora metodikutsu*

Suora metodinkutsu toimii kutsumalla suoraan tunnettua metodin nimeä, käyttämättä kuitenkaan normaalia metodikutsua.

```
1 Method metodi = Luokka.class.getMethod("metodi", String.class);
  metodi.invoke(olio, "teksti");
```

Ohjelma 4.4 *Metodikutsu käyttäen Javan Reflection -APIa*

Lookup on Java 7:n myötä tullut method handles -paketin osa, joka osaa etsiä luokalta tietyntyyppisen metodin. Lookup pystyy hakemaan tarkemmin halutun metodin luokalta, sillä sille on mahdollista määritellä metodin sallittu julkisuustyyppi ja se, onko metodi staattinen vai ei.

```
  MethodHandle methodHandle = MethodHandles.lookup()
2   .findVirtual(Luokka.class, "metodi", MethodType.methodType(String
      .class, String.class));
  methodHandle.invoke(olio, "teksti");
```

Ohjelma 4.5 *Metodikutsu käyttäen MethodHandles.Lookup -luokkaa*

Kun yllä olevat testit ajettiin 1 000 000 kertaa, kuluneet ajat olivat seuraavanlaiset:

Taulukko 4.1 *Reflektion nopeuden testiajo*

Kutsutapa	aika (ms)
1 000 000 kutsua käyttäen normaalia metodikutsua	5,75
1 000 000 kutsua käyttäen suoraa metodin kutsua	361,98
1 000 000 kutsua käyttäen lookupia metodin etsimiseen	1871,05

Kuten taulukosta 4.2 nähdään, normaali metodikutsu on ylivoimaisesti tehokkain ratkaisu. Tulokset ovat keskiarvot useammalta eri ajokerralta; testit ajettiin viiteen kertaan eri järjestyksessä, jotta Javan optimointi vaikuttaisi tuloksiin mahdollisimman vähän. Toinen testiajo suoritettiin kytkemällä Javan virtuaalikoneen optimointi pois päältä testien ajaksi. Tämä onnistui asettamalla Javalle käynnistysparametriksi `-Djava.compiler=NONE`.

Taulukko 4.2 *Reflektion nopeuden testiajo ilman optimointia*

Kutsutapa	aika (ms)
1 000 000 kutsua käyttäen normaalia metodikutsua	48,68
1 000 000 kutsua käyttäen suoraa metodin kutsua	2148,34
1 000 000 kutsua käyttäen lookupia metodin etsimiseen	42754,56

Toisesta optimoimattomasta testiajosta huomataan, että MethodHandeles.lookup -tavalla toteutettu haku on lähes 23 kertaa hitaampi. Tämä selittyy parhaiten sillä,

että ohjelma joutuu lataamaan välissä käyteyn MethodHandlesin logiikan joka kerta uudestaan. Normaali metodikutsu on 8 ja puoli kertaa hitaampi ja suora reflektiokutsu on noin 6 kertaa hitaampi. Tämä testitapaus ei voi täysin vastata oikeaa käyttötapausta, sillä optimointia tapahtuu aina ohjelman suorituksen aikana. Kuitenkin optimoimaton testaus antaa osviittaa siitä, miten uuden, aina erilaisen, kutsun ajaminen yhden kerran vaikuttaa ajoaikaan. Kun metodia kutsutaan vain kerran, sitä ei voida optimoida tulevia suorituskertoja varten.

5. TIETOKANTOJEN TALLENNUSRATKAISUT

5.1 Taustaa

Yleisesti ottaen tietokannat voidaan jakaa kahteen päätyyppiin; relaatiokantoihin ja ei-relaatiokantoihin. Relaatiokannoissa eri tiedot voivat viitata toisiinsa suoraan kannan sisällä, ja ei-relaatiokannoissa tätä ominaisuutta ei ole. Ei-relaatiokannat, eli NoSQL -kannat, ovat pääasiassa objekti- tai dokumenttikantoja, joihin joko tallennetaan paljon tietoa, tai jossa tiedonhaun nopeus on ensisijaista. Etenkin suuret verkkoyhtiöt kuten Google, Facebook, jne. käyttävät dokumenttikantoja valtavan tietomääränsä, niinsanotun big datan, tallentamiseen. Vaikka tieto sinänsä liittyy muuhun tietoon, sitä on kuitenkin niin valtavasti, että sen tallentamisen nopeus on tärkeää. [9]

Relaatiokannat ovat toinen vaihtoehto, jotka soveltuvat käytettäväksi suurimpaan osaan käyttötapauksista. Kannasta riippuen ne sopivat niin pieneen kuin suureen käyttöönkin, skaalautuen vaatimusten mukana. Relaatiokannat ovat hieman NoSQL -kantoja hitaampia, mutta relaatioiden avulla varmistavat kannan eheyden, jotta tietokantaan ei voida tallentaa muuhun järjestelmään nähden väärää tietoa, eikä oleellista muun järjestelmän vielä tarvitsemaa tietoa voida poistaa. Tämän kaiken voi NoSQL -kannassa varmistaa ohjelmallisesti, mutta se on työläämpää eikä yhtä varmaa kuin suoraan kantaan asetut rajoitukset.

Sekä SQL, että NoSQL -tietokannat tukevat jonkinlaista binääridatamuotoa, yleisesti BLOB (Binary Large Object). Tätä tallennusmuotoa käytettäessä tietokanta ei yritä tulkita dataa ollenkaan, vaan tallentaa sen kantaan täysin siinä muodossa kuin se vastaanottaa sen; yleensä muissa tapauksissa tietokannat asettavat kentille rajoitteita tai muokkaavat dataa omaan muotoonsa sopivammaksi. Koska tavu-

koodiksi käännetyt javaluokat eivät ole suoraan minkään tietokannan ymmärtämää muotoa, ne tallennetaan kantaan käyttäen tätä BLOB-muotoa.

Käytettävissä projektissa on valmiiksi SQL-tietokanta, josta löytyy tarvittavaa tietoa laskennan suorittamiseen, joten se on käytännössä ainoa vaihtoehto tietokannaksi. Myös sääntöjen suora liittyminen suorituksiin, arvoihin ja ohjauspolitiikoihin luo tietokantaan relaatioita, joten myös tältä osin relaatiokanta on parempi vaihtoehto.

5.2 Vaihtoehdot

Käytettävien luokkien tallentamiseen on kaksi eri vaihtoehtoa; luokat voidaan tallentaa joko kääntämättömänä koodina tai tavukoodina tietokantaan. Tavukoodina tallennetun luokan etuina on nopeampi lataus tietokannasta, sillä luokkaa ei tarvitse erikseen kääntää ennen käyttöönottoa. Koodina tallessa oleva luokka joudutaan kääntämään tavukoodiksi joka kerta kun se ladataan. Sinänsä javakoodina tallennettua luokkaa on helpompi muuttaa ajokertojen välillä, mutta mikäli koodi ladataan useammin kuin muutetaan on valmiiksi käännetty tavukoodi parempi ratkaisu. Tavukoodi vie binäärimuodossa tallennettuna myös vähemmän tilaa kuin koodi, joka tarvitsee kääntää ennen suoritusta.

Kumpikaan ratkaisu ei ole suoraan oikea, vaan käyttötapauksesta riippuva. Mikäli luokan toimintatapaa halutaan varmistaa, on selkokielinen koodi parempi tallennusratkaisu. Käännetty tavukoodi on myös mahdollista purkaa takaisin luottavaksi koodiksi, mutta sen rakenne ei pysy yhtä hyvänä ja luettavana.

Tässä toteutuksessa päädyttiin tallentamaan vain tavukoodi tietokantaan, koska se on mahdollista generoida sääntökielestä tarvittaessa uudestaan, eikä välimuotona olevalle javakoodille ole tarvetta.

5.3 Metatiedot

Jokainen tietokantaan tallennettu luokka tarvitsee mukaansa metatietoa. Näihin kuuluu luokan nimi, versio ja kuvaus. Lisäksi luokista tallennetaan mukaan niiden koodi sääntökielenä. Näitä tallennuksia varten käytetään tietokannan tavallisia tallennuskenttäformaatteja: tekstikentät nimelle ja kuvaukselle ja numerokenttä

versionumerolle. Sääntökielen koodille käytetään erikoispitkää tekstikenttää, jossa kapasiteetti ei lopu vaikka kielen koodi olisi hieman odotettua pidempikin.

Vaikka metatietoihin tallennetaan sääntökielellä oleva koodi, tietokanta ja järjestelmä ei kuitenkaan voi olla täysin varma, että tallennettu tavukoodi vastaa tätä koodia. Tämä tallennettu koodi on olemassa myöhempiä muokkaustarpeita varten, eikä niinkään laskennan toiminnan kannalta. Tämänlaisessa tapauksessa on oleellista ottaa tietoturva huomioon.

6. TIETOTURVANÄKÖKULMIA

6.1 Riskit

Riskienhallinta on oleellinen osa tämäntapaista projektia. Tietokantaan tallennetaan tavukoodia, josta ei voi suoraan lukea mitä se tekee. Tuntemattomasta tavukoodista ja sen suorittamisesta on tärkeää tehdä mahdollisimman turvallista. Potentiaalisia riskejä jotka voivat toteutua jos hyökkääjä pystyy ajamaan mielivaltaista koodia palvelimella on:

- Pääsy palvelimen tiedostoihin
- Muiden ohjelmien suorittaminen ohjelman oikeuksilla
- Pääsy tietokantaan
- Haittaohjelmien ja takaovien asentaminen

Jokainen näistä on mahdollinen riskitekijä, joka tulee ottaa huomioon kun järjestelmästä tehdään mahdollisimman tietoturvallinen.

6.2 Tietokanta

Tietokantaan tallennetaan käännetty tavukoodi, joten se on luonnollisesti yksi tietoturvariskeistä. Käyttöoikeudet käytettyyn tietokantaan rajataan vain niille käyttäjille, joiden tarvitsee päästä käsiksi siihen. Tavukoodin tallentamisen ja muokkaamisen oikeudet tulee rajata näistäkin vain mahdollisimman suppealle määrälle luotettuja tahoja. Lukuoikeudet voidaan jättää suuremmalle joukolle, mutta ei kuitenkaan kenelle tahansa. Kun tavukoodia tallennetaan tietokantaan on olemassa useampia kohtia joissa tietoturva on uhattuna. Tallentavassa ohjelmassa koodiin on mahdollista

päästä käsiksi ennen kuin sitä aletaan tallentamaan sekä myös tallentaessa tietokantaan. Käyttäjään tunnuksiin on pakko luottaa, nämä tulevat muualta järjestelmästä ja mikäli hyökkääjällä pääsy niihin niin tietokannassa ei ole olemassa keinoja joilla estää hyökkääjän pääsy tietoihin.

Tietokannan muutostapahtumista pidetään kirjaa ja muutoksen tekijä sekä aikaleima voidaan tallettaa logiin. Haavoittuvuus ohjelman suorituksen aikana on teoriassa mahdollista, mutta sekin vaatisi käyttöoikeudet suorittavalle koneelle, missa tapauksessa olisi varmasti helpompiakin keinoja muokata tietoja.

Suurin, perinteinen haavoittuvuus on SQL-injektiohyökkäys, jossa hyökkääjä suorittaa mielivaltaista tietokantakoodia palvelimella hyödyntäen vajavaisesti toteutettua SQL-kyselyä. Tämä on hyvin tiedostettu hyökkäys nykyaikana ja siihen osataan varautua.

6.3 Classloader

Jokainen luokka perii sen classloaderin, jota käytettiin luokan luomiseen. Tässä tapauksessa siis luokan tietokannasta lataava classloader voi asettaa rajoitukset käytettävissä oleville luokille. Whitelist olisi toimiva esimerkki, joka voisi antaa dynaamiselle luokalle pääsyn ainoastaan Javan perusmuuttujiin kuten String ja Integer. Mikäli haluttu toiminnallisuus vaatii suuren määrän eri luokkia, tämä lista saattaa paisua paljonkin, sekä jokainen uusi sallittu luokka lisää haavoittuvuuden todennäköisyyttä. Javassa on olemassa Byte Code Verifier, joka kuitenkin estää osan potentiaalisesti haitallisista käyttötavoista[10]. Tämä tavukoodivarmennin varmistaa, että koodi käsittelee osoittimia sen tyyppisinä kuin ne on määritelty, että koodi käsittelee vain osia, joihin sillä on oikeus ja varmistaa, että olioita käsitellään sen tyyppisinä kuin ne oikeasti ovat. Javasta kuitenkin löytyy, versiosta riippuen, eri keinoja paeta rajoitetusta classloaderista.

Tietoturva-aukkoja kuitenkin paikataan jatkuvasti, ja erityistä sallivaa whitelistiä käyttävästä classloaderista on huomattavasti vaikeampi murtautua ulos kuin kieltolistaa, blacklistia käyttävästä. On paljon helpomaa varmistaa muutaman luokan turvallisuus, kuin erikseen luetella kaikki ne luokat, joihin ei luoteta.

6.4 SecurityManager

SecurityManager on Javan yleinen turvallisuusmanageri, jolle voidaan asettaa rajoituksia mitä ohjelman suoritus saa tehdä. Mikäli jotain kiellettyä yritetään tehdä, heittää SecurityManager poikkeuksen ja estää toiminnan. SecurityManager on kuitenkin koko suoritukselle globaali, joten sitä ei voi asettaa pääohjelmalle vaaditun rajoitetussa moodissa. SecurityManagerin turvallisuuspolitiikka määrittelee saako suoritettava ohjelma suorittaa muun muassa seuraavan tyyppisiä operaatioita:

- Levyn luku- ja kirjoitusoperaatiot
- Reflektio-ominaisuuksien käyttö
- Verkon käyttö
- Tietoturva-asetusten käsittely
- Logittaminen
- Tietokantaoperaatiot
- SSL:n käyttö
- Uusien säikeiden luominen

SecurityManager mahdollistaa myös ohjelmien allekirjoituksen varmistamisen. Java Development Kitin mukana tulee jarsigner-ohjelma jolla on mahdollista digitaalisesti allekirjoittaa jar-tiedostot julkisen avaimen salauksella [11]. Tällä tavalla voidaan rajoittaa suoritettavan koodin oikeuksia myös sen perusteella, mikä allekirjoitus sillä on.

SecurityManagerin antamat mahdollisuudet suorituksen rajoittamiseen eivät kuitenkaan välttämättä riitä, sillä Javan historiassa on huomattavia tapauksia, joissa haittaohjelma on onnistunut suorittamaan koodia SecurityManagerin ohi. Esimerkiksi Oraclen vuonna 2013 korjaama haavoittuvuus[12] mahdollisti minkä tahansa ohjelman pääsevän käsiksi sun.org.mozilla.javascript.internal.GeneratedClassLoader -luokkaan käyttämällä reflektiota ja kytkemällä järjestelmän SecurityContextin pois

päältä. Tämän vuoksi SecurityManager ennemminkin täydentää tietoturvaa, kuin vastaa siitä kokonaisuudessaan. Javan haavoittuvuudet on korjattu uudemmissa versioissa, joten tieototurvaongelmia on kuitenkin todennäköisesti vähemmän.

6.5 Säikeet

Dynaamisen luokan suorittaminen omassa säikeessään ratkaisee SecurityManagerin ongelman. Säikeestä ei voi koskea muun ohjelman suoritukseen, sekä SecurityManager pystytään laittamaan päälle vain yhteen säikeeseen, jolloin se rajoittaa vain haluttua osaa ohjelman suorituksesta. Tämä yhdistettynä whitelistaavaan classloaderiin luo jo todella turvallisen hiekkalaatikon jossa dynaamisen luokan uskaltaa suorittaa. Säikeistäminen ratkaisee myös mahdollisen vahingossa tapahtuneen ohjelmointivirheen, jonka tuloksena on loputon silmukka. Ikuisesti ajavan säikeen tilaa voi tarkastella ulkopuolelta, sekä haluttaessa sen suoritus voidaan lopettaa. Esimerkiksi luokan suorituksen aikarajoitus on toimiva ratkaisu, kunhan rajoitus on tarpeeksi antelias, että pitkäksi venähtänyt normaalisuorituskin vähemmän tehokkaalla koneella voidaan suorittaa loppuun saakka.

Säikeessä suoritettavalla luokalla ei myös ole oikeutta aloittaa uusia säikeitä; muuten luokan olisi mahdollista avata teoreettisesti loputtomasti säikeitä. Tämä loppuisi kuitenkin Javan virtuaalikoneen heap-muistin loppumiseen ja koko ohjelman suorituksen kaatumiseen. Tämä ei ole toivottavaa, oli syynä sitten ohjelmointivirhe tai potentiaalinen hyökkäysyritys; uusien säikeiden luonnille ei kuitenkaan ole syytä.

Säikeiden luontikielto asettaa hieman rajoituksia sille, mitä luokilla on mahdollista tehdä, mutta rajoitus ei ole suuri. Suurin haitta koituu rinnakkaisten algoritmien suorittamisille. Mikäli säikeiden luonti halutaan mahdollistaa, täytyy se toteuttaa esimerkiksi niin kutsuttujen säiealtaiden kautta (ThreadPool). Yksi säieallas sisältää tietyn määrän säikeitä, joita on mahdollista käynnistää. Kun säikeet loppuvat altaasta, täytyy luokan odottaa, että jo varatun säikeen suoritus loppuu, ja se voidaan uudelleenkäyttää.

Jotta säikeesti ei olisi mahdollista murtautua ulos muuhun järjestelmään, täytyy seuraavien rajoitusten olla kiellettyjä SecurityManagerista:

- RuntimePermission("createClassLoader")
- RuntimePermission("accessClassInPackage.sun")
- RuntimePermission("setSecurityManager")
- ReflectPermission("suppressAccessChecks")
- FilePermission("<<ALL FILES>>", "write, execute")
- SecurityPermission("setPolicy")
- SecurityPermission("setProperty.package.access")

Mikäli yksikään näistä on päällä, on säikeeseen luodun prosessin mahdollista päästä käsiksi muuhun järjestelmään.

7. TOTEUTETTU JÄRJESTELMÄ

7.1 Luokkien luominen

Luokat luodaan perustoteutuksesta perityllä ClassLoader-toteutuksella, jolle annetaan tietokantayhteys, jonka avulla se pystyy lataamaan oikeaan pakettiin kuuluvat luokat tietokannasta niiden nimen perusteella. Järjestelmään kuuluu myös tietokantaan tallennettujen luokkien tilaa valvova muuttuja, jonka avulla saadaan tietää kun jokin luokka muuttuu. Tässä tapauksessa käytetty classloader joudutaan tuhoamaan ja korvaamaan uudella samanlaisella. Classloader ei pysty toimintatakuunsa vuoksi unohtamaan yhtään luomaansa luokkaa, vaan luo aina kaikki saman nimiset luokat identtisinä, jolloin tämä classloader joudutaan poistamaan ja korvaamaan uudella, kun luotava luokan toteutus muuttuu.

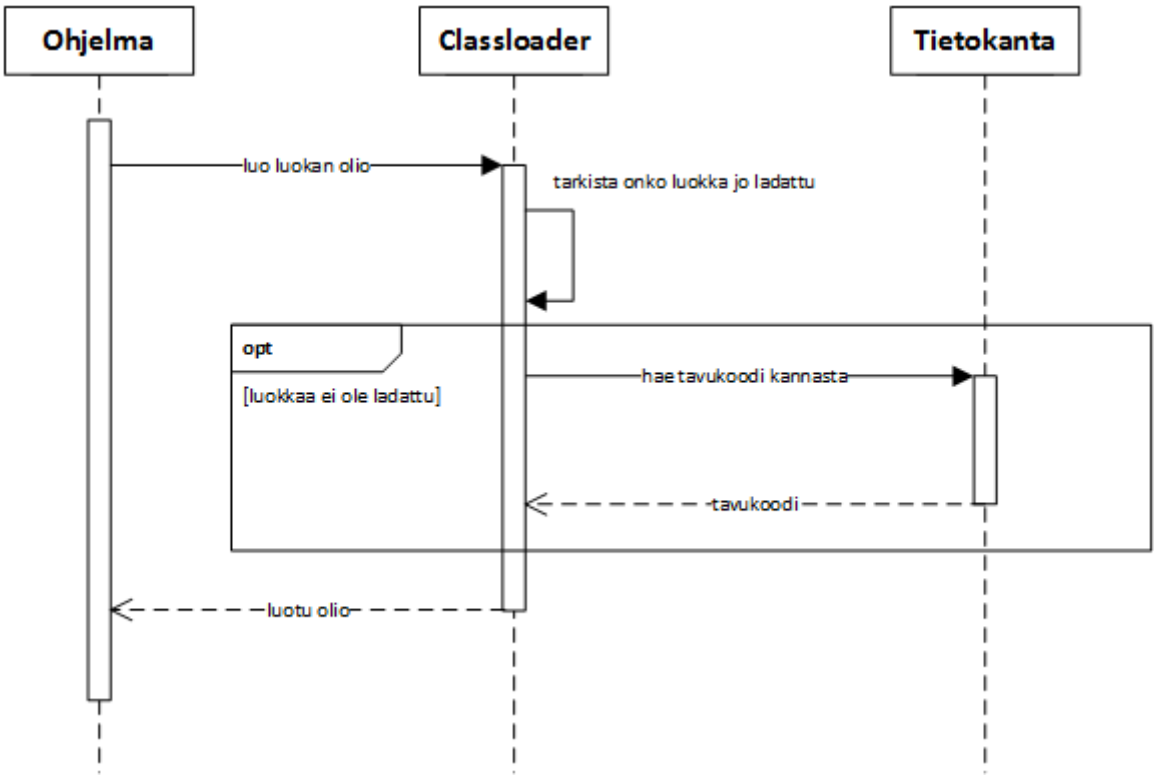
Luokka luodaan järjestelmään nimiformaatilla: {LuokanNimi}€{versionumero}. Luokan nimessä käytettävät merkit on rajoitettu merkkeihin a-ö, A-Ö sekä numeroihin, joten euromerkkiä ei ole mahdollista antaa nimelle normaalisti. Näin toteutettuna se toimii versionumeroeroittimena.

7.1.1 Versionti

Koska järjestelmän tulee tukea eri versioita saman nimisistä luokista, niihin ei voida suoraan viitata nimen perusteella. Classloaderit käyttävät kuitenkin luokista myös binäärinimiä, jotka muodostetaan lisäämällä €-merkki ja numerosarja luokan nimen perään, mikäli usaempi saman niminen luokka luotaisiin. Unicodemerkistön käyttö on sallittua luokkien ja metodien nimissä. Java käyttää omien saman nimisten tai geneeristen luokkien erottimeksi $ -merkkiä, mutta dollarimerkki on varattu symboli jota ei muuten ole mahdollista käyttää luokkien nimissä.

Javan classloaderit takaavat, että samalla classloaderilla luotu tietyn niminen luokka

Kuva 7.1 *Luokkien luontisekvenssi*



on aina identtinen kuin muut tällä classloaderilla luodut saman nimiset luokat. Lisäämällä erotin ja versionumero luokan perään on mahdollista pitää järjestelmässä useampia saman luokan versioita.

7.2 Luokan kääntäminen tavukoodiksi

Luokka käännetään tavukoodiksi sillä koneella, jolla sääntökieli kirjoitetaan. Se ensin käännetään sääntökielestä javakoodiksi, jonka jälkeen tämä javakoodi annetaan eteenpäin javac -kääntäjälle. Kääntäjä tuottaa tavukoodista koostuvan .class -tiedoston. Tämä luokkatiedosto lähetetään eteenpäin palvelimelle osana viestiä, joka sisältää myös luokan metatiedot, kuten nimen ja parametrit. Palvelin käsittelee saapuvan viestin ja tallentaa luokan ja tiedot tietokantaan.

7.3 Luokkien käyttäminen

Uusien dynaamisesti luotujen luokkien käyttämiseen on Javassa käytännössä kaksi eri tapaa; reflektio tai ennaltamäärätyn rajapinnan toteuttaminen.

Reflektion käyttäminen dynaamisesti luotujen luokkien ajamiseen todettiin olevan epävarmempaa ja marginaalisesti hitaampaa kuin rajapinnan käyttämisen. Mikäli tulevaisuudessa ajettavien luokkien käyttöön tarvitsee tehdä muutoksia, rajapinta takaa sen, että luokkien luominenkin päivittyy oikein. Reflektion avulla metodien kutsumista ei voi taata, että vanhalla versiolla luotu luokka toteuttaa enää metodia, jolloin sen kutsuminen epäonnistuu. Samassa tapauksessa päivitetyn rajapinnan kanssa saadaan jo luokan luomisvaiheessa tietää, että se on vanhenutunut eikä sitä mahdollisesti pystytä enää käyttämään.

Rajapinnan määrittelystä ja käyttämisestä tulee marginaalisesti lisätyötä sekä javakoodin generointiin sääntökielestä sekä luokkia luovaan classloaderiin, mutta toimintavarmuus sekä testattavuuden parantuminen on täysin sen arvoista. Toteutettuun rajapintaan jätettiin kuitenkin jonkin verran dynaamisuutta auki käyttäen Object -tyyppisiä parametreja ja paluuarvoja, joiden oikea tyyppi päätellään kontekstista.

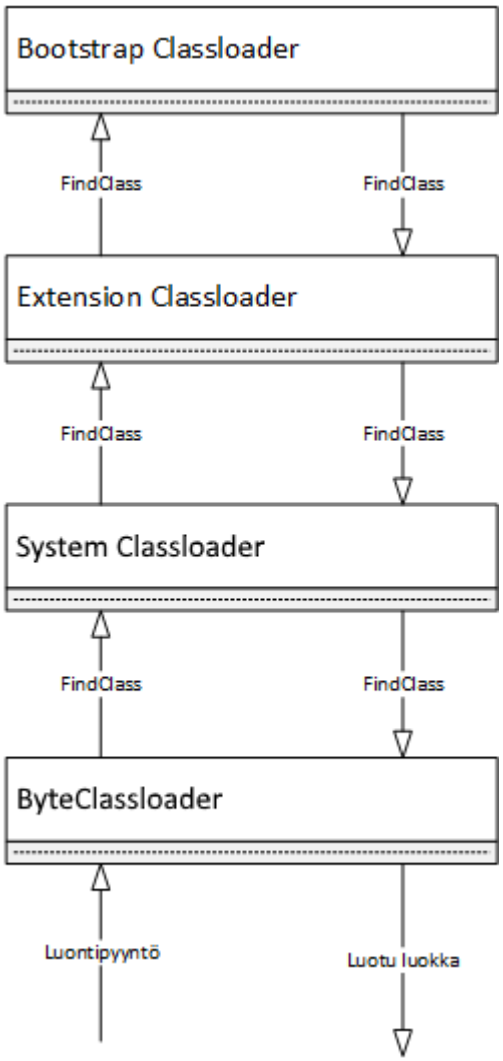
Näistä seikoista johtuen toteutuksessa päädyttiin käyttämään vain ennalta määriteltyä rajapintaa. Reflektiomaisiksi ominaisuuksiksi kuitenkin jäi käyttöön luokan tyypin päättely käyttäen Javan instanceof -kutsua, jolla voi kysyä luokalta, edustaako se kysyttyä luokkaa.

7.4 Tallennusratkaisu

Perinteisesti tämänlaisessa järjestelmässä olisi selkeämpää käyttää kovalevyä luokkatietojen tallentamiseen, mutta tässä tapauksessa se ei onnistunut. Järjestelmä on hajautettu useammalle noodille, jotka kaikki käyttävät yhteistä tietokantaa. Tietokanta tässä tapauksessa mahdollistaa helposti sen, että usemmalta noodilta voidaan olla yhteydessä siihen sekä luokkia voidaan lukea ja kirjoittaa yhtaikaa. Valintaan vaikutti myös se, että projektissa on valmiiksi jo tietokanta käytössä, jolloin sen käyttöönotosta ei aiheutunut ylimääräistä työtä.

Luokkatavukoodin tallentaminen tehdään SQL -tietokantaan, jossa ne identifioidaan nimen perusteella. Itse tavukoodi on BLOB-tyyppisessä kentässä tallessa.

Kuva 7.2 *Toteutettu classloaderhierarkia*



Versioinnin ja luokkien käyttämien parametrien tallentaminen oli huomattavasti suurempi ongelma.

7.5 Rajapinta

Dynaamisten luokkien vastuulla ohjelman suorituksessa on ohjaussuositusten luominen, ne vastaanottavat joko muualla järjestelmässä määriteltyjä arvoja tai mittausarvoja ja tuottavat näiden perusteella ohjaussuosituksia. Sekä saapuvat arvot, että tuotettavat tulokset käsitellään yksinkertaistetusti "Parametreina", joita käsitellään seuraavassa osiossa.

Jotta toteutettu järjestelmä pysyisi mahdellisimman selkeänä ja virtaviivaisena, dynaamisesti luodut luokat toteuttavat vain hyvin pelkistetyn rajapinnan. Luokille on mahdollista antaa lista parametreja, joista luokka saa luettua suorittamista varten tarvitsemansa tiedot. Luokilla on suorittamista varten oma metodinsa, jolla luokka suorittaa laskentansa. Luokka voi muokata sille annettun parametrilistan sisältöä, ja tämä on pääsääntöinen laskennan suorituskeino. Tämän vuoksi luokille annetut parametrit voidaan lukea omalla parametrilistan-hakumetodilla.

setParameters asettaa laskentaparametrit

getParameters lukee luokassa tällä hetkellä olevat parametrit

suorita suorittaa luokan laskennan

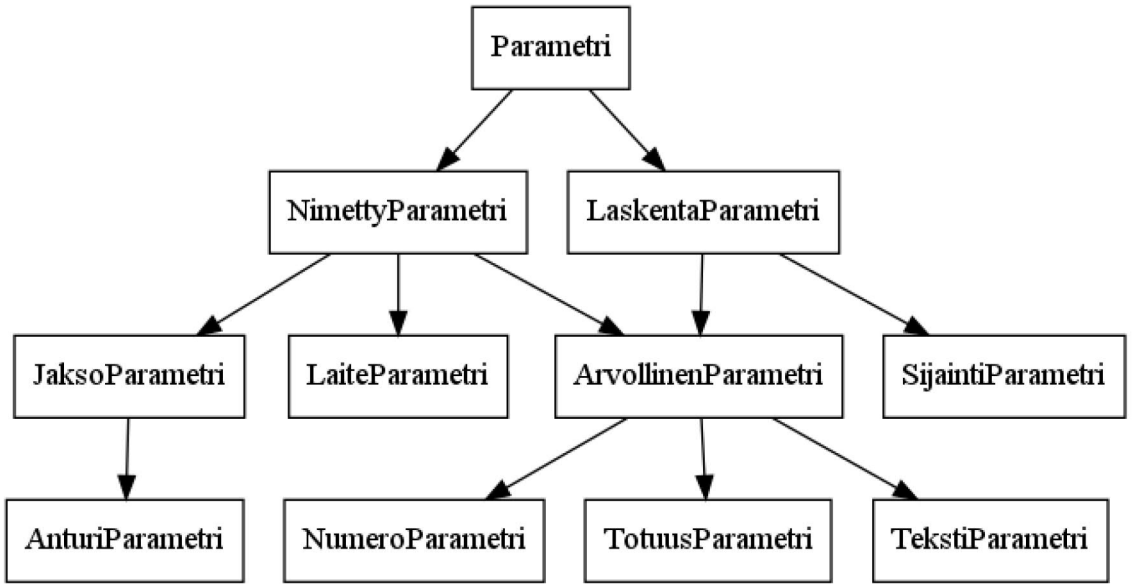
Rajapinta mahdollistaa kaikki järjestelmän tarvitsemat operaatiot; tulevaisuudessa on mahdollista laajentaa rajapinnan mahdollisuuksia, mutta toteutuksen tähän vaiheeseen tämä on riittävä.

7.6 Parametrit

Koska luokat ajetaan niille määriteltyjen parametrien perusteella, myös parametrit tallennetaan tietokantaan, josta ne haetaan ajon aikana. Nämä on tallennettu relaatiokantaan aggregaationa jokaiselle tallennetulle suoritustyypille. Koska näille suoritustyyppille tiedetään käytettävät parametrit etukäteen ennen ajoa, ne pystytään hakemaan ennen ajoa tietokannasta. Tällä saavutetaan huomattavia tehokkuushyötyjä verrattuna siihen, että tarvittavat tiedot haettaisiin laiskasti luokan suorituksen aikana, sillä nyt selvitään vain yhdellä kyselyllä.

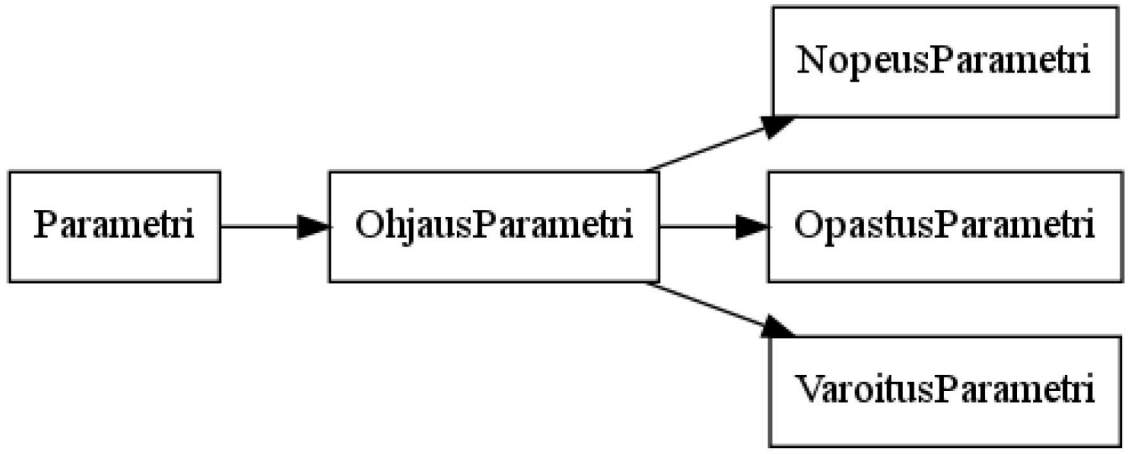
Toteutuksessa parametrit toteuttavat hyvin yksinkertaisen rajapinnan, joka mahdollistaa vain arvon lukemisen tai asettamisen. Parametrit voivat olla tapauksesta riippuen joko ohjaus- tai arvoparametreja. Eli toisin sanoen parametrit ovat joko in- tai out -parametreja. Toteutus sallii myös in-out -parametrien käytön, mutta näiden käyttöä ei ole vielä nähty tarpeelliseksi. Kuvassa 7.3 on syöteparametrien luokkahierarkia.

Kuva 7.3 *Syöteparemetrit*



Parametreja käsitellään Object -tyyppisenä listana, jonka yksilöt kuitenkin muutetaan tyyppimuunnoksella oikeaksi luokaksi listaa luettaessa ja käytettäessä. Parametrin oikea tyyppi on oleellista tietää, jotta sen antamat arvot osataan käsitellä oikein. Syöttö- ja ulostuloparametrit toteuttavat eri rajapinnat. Syöttäparametrit toteuttavat Parametri -rajapinnan, ja ulostuloparametrit toteuttavat Ohjausparametri -rajapinnan. Sisääntuloparametrit jaetaan myös kahteen eri tyyppiin, onko parametrilla nimeä vai onko se dynaaminen käytön aikana luotu. Nimetyt parametrit ovat sellaisia, joihin on mahdollista viitata sääntökielessä suoraan nimeltä. Ainoa laskentaparametri jolle ei ole nimeä, on SijaintiParametri, joka nimensä mukaan kertoo sijainnin ja niitä voi olla vain yksi per suorituskerta.

Kuva 7.4 *Ohjausparametrit*



Ohjausparametrit jaetaan sen mukaan kolmeen eri tyyppiin, että minkälaisen ohjauksen ne tuottavat. Samalla ohjaukset kertovat minkätyyppiselle laitteelle ohjaus on; infonäyttöä ei voi ohjata nopeusrajoituksella. Ohjausparametrien luokkahierarkia on esitetty oheisessa kuvassa 7.4.

8. JÄRJESTELMÄN KÄYTTÖ JA TOIMINTA

8.1 Käytön flow

Järjestelmän käyttö alkaa sääntöjen suunnittelijalta omalla työasemallaan. Käyttäjä ottaa yhteyden ohjaussuosituslaskennan käyttöliittymästä palvelimelle. Hän luo säännön kirjoittamalla sen sääntötyökalussa sääntökielieditoriin ja valitsee sen käyttämät parametrit. Kun sääntö on valmis, käyttöliittymä kääntää sääntökielikoodin Javan tavukoodiksi ja lähettää sen palvelimelle jolle on sillä hetkellä yhdistettynä.

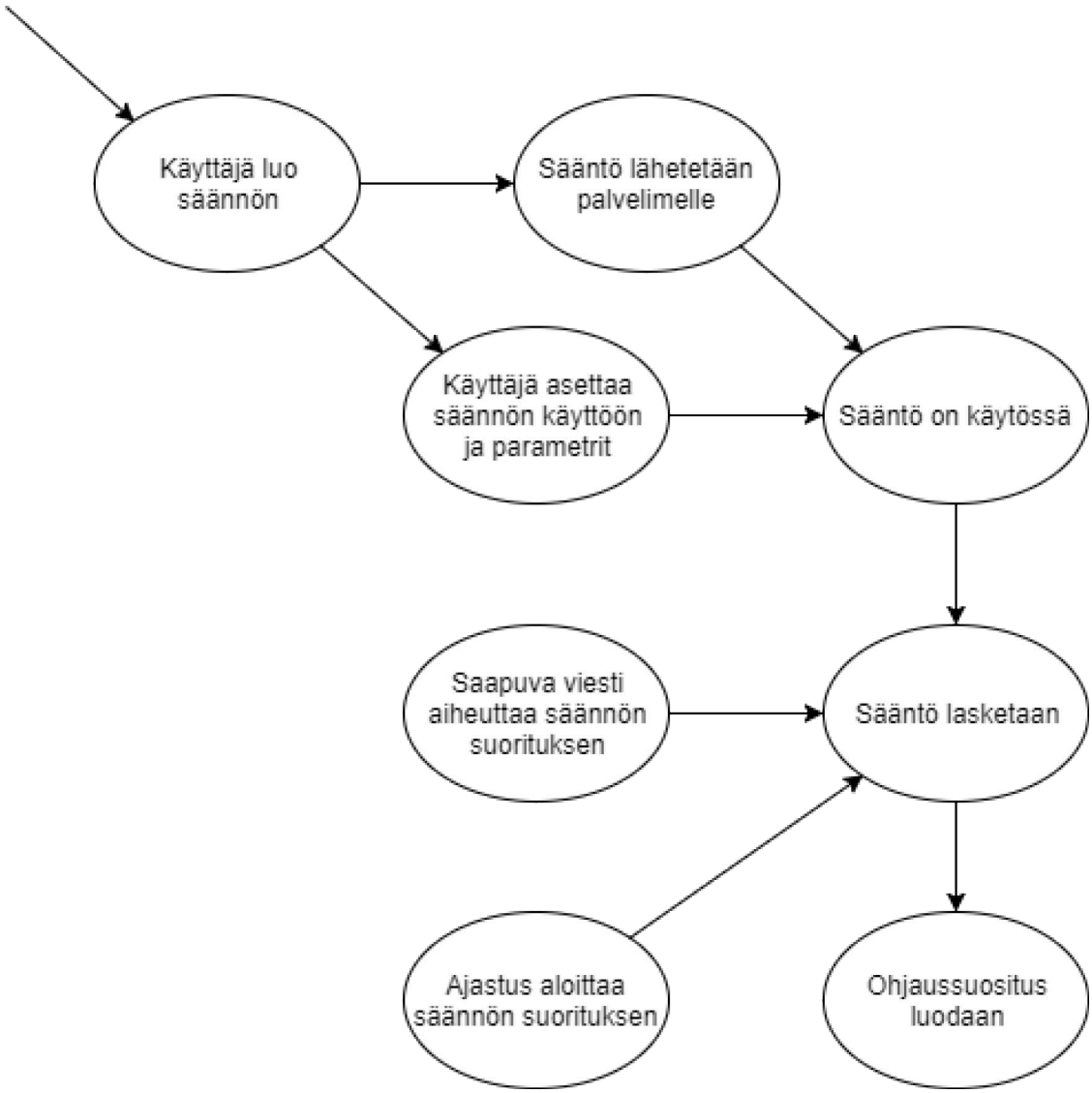
Palvelin vastaanottaa tavukoodin ja tallentaa sen tietokantaan. Samassa viestissä mukana viedään myös kääntämätön sääntökoodi, säännön nimi, parametrit ja metatiedot, jotka myös tallennetaan tietokantaan.

Käyttäjä voi asettaa säännön suoritettavan tiettyihin ohjauspolitiikoihin ja ohjausjaksoihin, mutta tämä toiminnallisuus ei ole tämän diplomityön kannalta oleellinen.

Normaalissa tapauksessa kaikki sääntöjen ilmeentymät suoritetaan kerran minuutissa niillä parametreilla, jotka on asetettu toteutettavaksi. Yksi sääntö voidaan suorittaa useita kertoja eri parametreilla, esimerkiksi samalla moottoritiellä voi olla monta muuttuvaa nopeusrajoituskylttiä, joiden syötelaitteet ja nopeusrajoituskyltit vain muuttuvat. Tässä tapauksessa sääntö on sama, mutta parametrit vaihtuvat. Säännöt voidaan myös suorittaa, kun järjestelmä vastaanottaa ei-mittausdata kuten tielle asetettuja häiriömerkintöjä. Tässä tapauksessa vain häiriömerkinnän kannalta oleelliset säännöt suoritetaan.

Kun säännöt on suoritettu, niiden tuottamat ohjaussuositukset kerätään, tarkastellaan mitkä niistä muuttuivat edelliseen suorituskertaan verrattuna ja lähetetään nämä muuttuneet eteenpäin järjestelmässä. Nämä suositukset mahdollisesti päätyvät ohjauksiksi teiden varsille.

Kuva 8.1 *Järjestelmän ajon flow*



8.2 Tekninen flow

Teknisesti flow on lähes sama kuin käyttäjänkin näkökulmasta. Järjestelmä voi sisältää määrittelemättömän määrän sääntöjä, jotka assosioidaan eri ohjauspolitiikkoihin. Kun sääntö tallennetaan järjestelmään, sen mukana tulevat parametrit tallennetaan omiin tauluihinsa ja näistä luodaan relaatio sääntöön. Kun sääntö assosioidaan johonkin ohjauspolitiikkaan, tästä luodaan oma relaatio ohjauspolitiikan, -jakson ja säännön välille. Tälle esiintymälle luodaan myös omat arvot säännön parametreille. Ne voidaan asettaa joko staattisiksi tälle esiintymälle, tai luettavaksi jostain saapuvasta datasta laitteen uniikin identifioijan perusteella.

Sääntöjen suoritus tapahtuu kerran minuutissa tai kun järjestelmä vastaanottaa syötteen, joka laukaisee sääntöjen suorittamisen. Suoritukset tapahtuvat rinnakkaisissa säikeissä käyttäen ThreadPool -palvelua, jolle on varattu maksimimäärä säikeitä, jotka voivat olla kerrallaan käytössä sääntöjen suorttamiseen. Tämä maksimimäärä on asetattavissa järjestelmän konfiguraatiossa.

Kun kaikki säännöt on suoritettu, niistä luetaan luodut ohjaukset ja nämä luodut ohjaukset kerätään talteen. Ohjaukset kootaan ohjaussuosituksiksi jotka lähetetään eteenpäin järjestelmässä seuraavalle tasolle, joka vastaa siitä, mitä ohjaussuosituksille tapahtuu. Järjestelmä tuottaa vain suosituksia, joten niiden käyttö ei ole varmaa, vaan seuraava taso päättää oman automaation tai ihmiskäyttäjän mielipiteen perusteella, minkälaisia ohjauksia lopulta päätyy perille asti.

9. TULEVAISUUS

Järjestelmän kehitystä tullaan jatkamaan vielä tulevaisuudessa. Todennäköisimmin kehitys tulee keskittymään uudentyyppisten syötteiden lisäämiselle ja ajettavien sääntöjen konfiguroitavuuden tehostamiseen. Ohjaussuosituslaskennan pohjan oletetaan olevan jo riittävän korkealla tasolla, että sitä ei tarvitse enää muuttaa.

Dynaamisiin luokkiin tuskin tullaan tekemään suuria muutoksia ilman kovin suuria perusteluita. Järjestelmän kannalta oleellinen asia, eli luokkien toteuttama rajapinta, tulee säilyä samanlaisena mikäli halutaan, että myös vanhat, aiemmin luodut säännöt toimivat. Mikäli sääntöjen rajapintaa muutetaan, vanhojen sääntöjen allekirjoitus ei enää täsmää uuden rajapinnan kanssa eikä classloader pysty lataamaan niitä toteuttamaan vanhaa rajapintaa. Tässä tapauksessa reflektio toimisi paremmin kuin rajapinta.

On kuitenkin mahdollista, vaikkakin työlästä, toteuttaa järjestelmä, jossa rajapinnan päivittyessä säännöille ja parametreille tallennetaan myös rajapintaversio, jota ne käyttävät. Tällöin sääntöä suoritettaessa järjestelmän tulee valita oikea rajapinta, jonka sääntö tarvitsee toimiakseen. Yksikin virhe latauksessa lopettaa säännön suorittamisen kerralla.

10. YHTEENVETO

Yhteenvetona dynaamisten luokkien luominen tietokannasta on erityisen hyödyllistä, kun ohjelman ajon aikana käyttämät luokat voivat muuttua ja järjestelmä on hajautettu useammalle noodile. Vastaavan järjestelmän toteuttaminen käyttämällä staattisia tietorakenteita vaatisi huomattavasti monimutkaisemman toteutuksen, jotta kaikki tarvittava logiikka olisi mahdollista. Tämä asettaisi varsinkin tietokannan suunnittelulle suuria haasteita. Myös myöhemmin toteutettava logiikan laajentaminen ja monimutkaistaminen käy todennäköisesti työläämmäksi.

Kun järjestelmä mahdollistaa dynaamisesti ladattavat luokat, jotka voivat toimia omassa rajoitetussa ympäristössä, saaden sisääntuloparametrit sekä antaen ulostuloparametrit, järjestelmästä tulee huomattavasti adaptiivisempi. Tämä kuitenkin asettaa tietoturvan vielä tavallista enemmän tarkastelun alle. Koska suoritettavan koodin sisällöstä ei ole täyttä varmuutta, täytyy sen oikeuksia ympäröivään järjestelmääään rajoittaa. Levyn luku- ja kirjoitusoperaatiot sekä käyttöjärjestelmäkutsut tulee olla kiellettyjä.

Luokkien dynaamisuus vaikeuttaa myös niiden testaukselle. Luokkien toimintaa on mahdollista testata ja varmistaa kun ne luodaan, mutta tämä testausympäristö ei kuitenkaan täysin vastaa oikeasti suorittavaa ympäristöä. Toisaalta luokkien yksikkötestaaminen erillään ympäristöstä on positiivinen seikka, sillä suorittavan hiekkalaatikon ominaisuuksien pitäisi olla sisään- ja ulostuloparametrien osalta täysin tunnettu. Tämä tarkoittaa, että integraatio- ja systeemitestaukselle ei ole tarvetta luokkien osalta.

Tietokanta ei ole luokille kaikista tavanomaisin ympäristö, mutta toimii tässä tapauksessa hyvin, sillä suorittavia noodeja on olemassa useita, joista jokaisella tulee olla luku- ja kirjoitusoikeus luokkia sisältäviin tauluihin. Tiedostojärjestelmä ei selviytyisi versioinnista ja transaktioiden eheydestä yhtä hyvin. Tietokanta antaa tä-

män varmistuksen suoraan ja varmuus on tärkeää tämän tapaisessa järjestelmässä.

T-LOIK:n suosituslaskennan tapaukseen dynaamiset luokat sopivat todella hyvin, ympäristön ja suoritettavan logiikan monimutkaisuuden ansiosta.

LÄHTEET

[1] H. Saarinen ja T. Laine. *Tieliikennekeskuksen operatiivisen toiminnan kehittäminen T-LOIK:n tuomat mahdollisuudet huomioiden.* Saatavilla: http://www.doria.fi/handle/10024/121655. 2012. ISBN: 978-952-255-219-8.

[2] Liikennevirasto. *Vaihtuvan ohjausjärjestelmän ohjauspolitiikan laadinta.* Saatavilla: http://www2.liikennevirasto.fi/julkaisut/pdf8/lo_2014-19_vaihtuvan_ohjausjarjestelman_web.pdf. Liikennevirasto, 2014.

[3] J. Peltonen. "Liikenneohjausjärjestelmän Sääntökieli". Saatavilla: http://urn.fi/URN:NBN:fi:tty-201703131151. Tutkielma. 2017.

[4] *Xtend.* WWW-Sivu, Viitattu 12.11.2017 http://www.eclipse.org/xtend/. 2017.

[5] *Xtext.* WWW-Sivu, Viitattu 12.11.2017 https://www.eclipse.org/Xtext/. 2017.

[6] T. L. ja Frank Yellin ja Gilad Bracha ja Alex Buckley. *The Java Virtual Machine Specification.* Saatavilla: https://docs.oracle.com/javase/specs/jls/se8/jls8.pdf. Oracle, 2015.

[7] *GCJ.* WWW-Sivu, Viitattu 12.11.2017 https://web.archive.org/web/20070509055923/http://gcc.gnu.org/java/. 2017.

[8] F. C. T. ja M. J. Santana ja R. H. C. Santana ja S. M. Bruschi ja J. C. Estrella. "WSBCL: Web Services Based Classloader" (2011). IEEE: 10.1109/WETICE.2011.23.

[9] A. j.V.L.j.D.e. a. Reniers V. ja Rafique. *J Internet Serv Appl.* Saatavilla: https://doi.org/10.1186/s13174-016-0052-x. 2017.

[10] Oracle. WWW-Sivu, Viitattu 12.11.2017, http://www.oracle.com/technetwork/java/security-136118.html. 2017.

[11] Oracle. WWW-Sivu, Viitattu 12.11.2017, https://docs.oracle.com/javase/7/docs/technotes/tools/windows/jarsigner.html. 2017.

[12] Oracle. *Oracle Security Alert for CVE-2013-0422*. Saatavilla: `https://www.oracle.com/technetwork/topics/security/alert-cve-2013-0422-1896849.html`. 2013.